IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 5, 2020, accepted June 14, 2020, date of publication July 6, 2020, date of current version July 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3007338

CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure

KENNEDY A. TORKURA, MUHAMMAD I. H. SUKMANA, FENG CHENG, AND CHRISTOPH MEINEL
Hasso-Plattner-Institute for Digital Engineering, University of Potsdam, 14469 Potsdam, Germany
Corresponding author: Kennedy A. Torkura (kennedy.torkura@hpi.de)

ABSTRACT Most cyber-attacks and data breaches in cloud infrastructure are due to human errors and misconfiguration vulnerabilities. Cloud customer-centric tools are imperative for mitigating these issues, however existing cloud security models are largely unable to tackle these security challenges. Therefore, novel security mechanisms are imperative, we propose Risk-driven Fault Injection (RDFI) techniques to address these challenges. RDFI applies the *principles of chaos engineering* to cloud security and leverages feedback loops to *execute*, *monitor*, *analyze* and *plan* security fault injection campaigns, based on a *knowledge-base*. The knowledge-base consists of fault models designed from secure baselines, cloud security best practices and observations derived during iterative fault injection campaigns. These observations are helpful for identifying vulnerabilities while verifying the *correctness* of security attributes (*integrity, confidentiality and availability*). Furthermore, RDFI proactively supports risk analysis and security hardening efforts by sharing security information with security mechanisms. We have designed and implemented the RDFI strategies including various chaos engineering algorithms as a software tool: *CloudStrike*. Several evaluations have been conducted with CloudStrike against infrastructure deployed on two major public cloud infrastructure: Amazon Web Services and Google Cloud Platform. The time performance linearly increases, proportional to increasing attack rates. Also, the analysis of vulnerabilities detected via security fault injection has been used to harden the security of cloud resources to demonstrate the effectiveness of the security information provided by CloudStrike. Therefore, we opine that our approaches are suitable for overcoming contemporary cloud security issues.

INDEX TERMS Cloud security, security chaos engineering, resilient architectures, security risk assessment.

I. INTRODUCTION

Cyber-attacks against Infrastructure as a Service (IaaS) cloud platforms have increased in recent years, mostly exploiting *configuration* vulnerabilities. These types of vulnerabilities include misconfigured Access Control Policies (ACP), over-privileged users and lack of audit logging. Consequently, the Cloud Security Alliance (CSA) Top Cloud Computing Threats 2019 report [1] identified *data breaches due to misconfiguration* and *inadequate change control* as the top 2, most severe cloud security threats. Similarly, the Ponemon Institute's Data Breach Report 2019, disclosed that 49 % of breaches are caused by system glitches and human errors [2]. The key takeaway from these reports and similar research is that cloud customers are the weakest link in the cloud ecosystem [3]. Furthermore, while CSPs fulfill their responsibilities as specified in the Shared Security Responsibility Model (SSRM), most cloud customers are yet to implement the requirements of the SSRM. There are several reasons for this including lack of efficient, customer-centric tools [4], wide cloud skills gap [1], [5] and increasing complexity of cloud services. Some of these challenges can be resolved by evolving proactive customer-centric security mechanisms [1].

We tackle the above-mentioned security challenges with a novel concept - Risk Driven Fault Injection (RDFI), a unique application of *chaos engineering* [16], [17] to cyber- security. RDFI extends the principles of chaos engineering[1] to cloud security to gain security benefits, additional to the already established resiliency benefits. The *state-of-the-art*

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Imran Tariq.

[1] https://principlesofchaos.org/

TABLE 1. Chaos engineering frameworks.

Framework	Target Stack	Resiliency Attributes	Application Layer
Chaos Kong [6]	AWS Regions	availability (non-security)	cloud network
Chaos Gorilla [7]	AWS Availability Zones	availability (non-security)	cloud network
Chaos Monkey [8]	AWS Availability Zones	availability (non-security)	cloud instances (VMs)
Chaos Monkey for Spring Boot [9]	Spring Boot Applications	availability (non-security e.g. latency, terminations)	Java applications (e.g. REST, inter-service calls)
Royal Chaos [10]	Java applications	availability (non-security)	JVM
Chaos Toolkit [11]	AWS, Azure, Google & Kubernetes	availability (non-security)	cloud & kubernetes network
PowerfulSeal [12]	Kubernetes	availability (non-security)	kubernetes network (e.g. pods, microservices)
ChaosMesh [13]	Kubernetes	availability (non-security)	kubernetes network
ChaoSlingr [14]	AWS	security (confidentiality, integrity & availability)	cloud services (e.g. S3, IAM)
CloudStrike [15]	AWS & GCP	security (confidentiality, integrity & availability)	cloud services (e.g. S3, IAM)

chaos engineering techniques inject *faults* into software systems to detect availability issues e.g latency. Subsequently, these issues are resolved to improve system resilience thereby enabling confidence in the system's capability to withstand turbulence [17]. In general, implemented resiliency patterns e.g. *timeouts, retries, and fallbacks* are important for chaos engineering experiments, given the provision of clear feedback information about system behavior [18], [19]. These feedback are indicative of faults, thereby providing opportunities for mitigation. However, these resiliency strategies are not designed to improve security, rather, they are designed to tackle availability challenges.

Since faults and failures could also impact security, it makes sense to derive similar resiliency strategies for security. Table 1 is a summary of some notable chaos engineering tools, we can notice the diversity of applications i.e. across several abstraction layers, but most tools focus on solving issues related to non-security availability challenges. Hence, we aim at devising ways for transferring the gains of resiliency patterns to security. Conversely, we propose the notion of RDFI, for injecting security faults, that detect security vulnerabilities i.e. failures that impact confidentiality, availability and integrity. Similar to the feedback loops employed for non-security faults, we propose an adaptation of the Monitor Analyze Plan Execute over-a-shared Knowledge-base (MAPE-K) feedback loop [20], which has been popularly employed in complex, autonomous computing. Our adaptation provides an effective model for automating the process of acquiring and transferring security information gained via security fault injection to security mechanisms e.g. firewalls, for remediation. These faults are codified as hypotheses to verify the correctness of security tools, controls and attributes. For example, a hypothesis might be: *is the principle of least privilege well configured for Amazon Web Services (AWS) S3 bucket XYZ?*

RDFI is implemented in *CloudStrike* [15], a cloud security system that implements chaos engineering principles. We extended our initial work in [15], by implementing security *fault models* drawn from secure baselines, industry standard best practices e.g. the Centre for Internet Security (CIS) benchmarks for Cloud Service Provider (CSP) [21], [22] and the CSA cloud penetration testing playbook [23]. These guidelines provide guidance for cloud security, which we leveraged to construct test suites for injecting security faults. Additionally, in order to achieve non-random, sequential exploration of the *fault space*, (attack surface) *cloud attack graphs* were employed. To the best of our knowledge, there is no other work that injects security faults using similar techniques.

Contributions: In an earlier *work-in-progress* paper [15], we proposed basic concepts for applying the principles of chaos engineering to cloud security. In this article, we have extended the work in the following ways:

- We establish the relationship between chaos engineering and related concepts: dependability, security and resiliency thereby demonstrating that security can be practically expressed as an attribute of resiliency (Section II-B).
- We propose the notion of RDFI, which applies security risk paradigms e.g. attack graphs and vulnerability scoring, to chaos engineering (Section III).
- We propose the RDFI Feedback Loop (adapted from the MAPE-K model [20]), as a model for automating the transfer of security information gained via Security Chaos Engineering (SCE) to cyber-security controls and mechanisms (Section III-A).
- Several security fault models that aid in detecting security vulnerabilities in cloud infrastructure are proposed (Section III-C).

- We implemented our concepts as a software tool: CloudStrike (Section IV), and conducted extensive experiments using realistic, state-of-the-art attacks against two major CSPs: AWS and Google Cloud Platform (GCP) (Section V).

The rest of this paper is structured as follows, in the next section, we introduce a running example to consolidate our concepts, and thereafter establish the relationship between chaos engineering, dependability, security and resiliency. In Section III, we present the RDFI feedback loop, RDFI, our fault models and how the principles of chaos engineering are applied in RDFI. The design and implementation of CloudStrike is highlighted in Section IV, while results of our evaluation are in Section V. Related works are presented in Section VI, while interesting next steps are highlighted in Section VII. The paper is concluded in Section VIII.

II. SECURITY CHAOS ENGINEERING

This Section discusses important concepts around SCE: the application of chaos engineering concepts to cyber-security. According to Rinehart [24], *SCE is identification of security control failures through proactive experimentation to build confidence in the system's ability to defend against malicious conditions in production.* Therefore, a running example based on a real data breach incident is first introduced, then the relationship between chaos engineering, dependability, security and resiliency is highlighted. Finally, our methodology for ensuring safe experiments in presented.

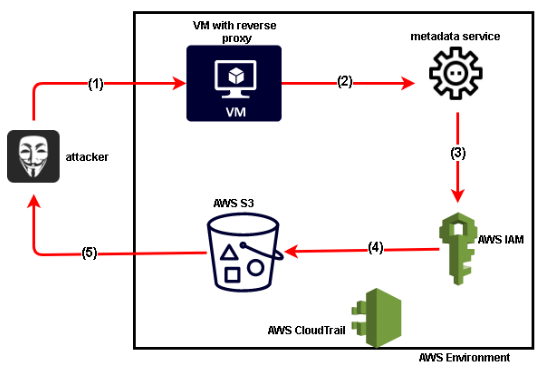


FIGURE 1. Running example- an illustration of the capital one data breach.

A. RUNNING EXAMPLE - THE CAPITAL ONE DATA BREACH

To provide a concrete illustration of contemporary cloud security issues, we would use the Capital One data breach [25], as a running example. This data breach occurred due to several attacks against Capital One's AWS infrastructure, Figure 1 is a summary of the attack scenario. The initial *entry point* (**EP01**) was a misconfigured reverse proxy, that the attacker identified and leveraged to gain access to an Elastic Computing Cloud (EC2) VM (*Step 1*), where the reverse proxy server was hosted. Having gained an initial *foothold*, the attacker executed a Server Side Request Forgery (SSRF) attack against the metadata server (*Step 2*), to obtain valid and extensive permissions. The metadata server in turn requests for permissions from the AWS Identity and Access Management (IAM), as defined in the profile access control policy (*Step 3*). These permissions are quite broad, granting access to the entire AWS Simple Storage Service (S3) i.e. the VM (including any user inheriting the permissions scoped within the VM) can make root-level requests against all assets within the AWS S3. Therefore the attacker inherits these privileges (*Step 4*) (**EP02**), by virtue of taking control of the VM. Thereafter, the attacker retrieves several critical information from the S3 bucket (*Step 5*) e.g. customers' email addresses, social security numbers and credit card information (**EP03**). In the above scenario, we notice several security issues: (a)misconfigured reverse proxy (**EP01**), (b) over-privileged profile policy, that does not satisfy *the principle of least privilege* (**EP02**) (c) massive ex-filtration of sensitive data from the S3 bucket without triggering alarms (**EP03**). These security issues are due to misconfigured cloud assets and ought to be prevented by implementing security controls.

B. CHAOS ENGINEERING

Chaos engineering [16], [18] has emerged as a discipline to enable *resiliency* in the cloud. According to Basiri et.al [16], chaos engineering *is the discipline of experimenting on a distributed system in order to build confidence in its capability to withstand turbulent conditions in production*. At the core of chaos engineering is the idea of conducting experiments to either affirm or disprove hypotheses. Here, a hypothesis refers to an *expected* or *assumed* behavior of a system, under specific scenarios. During chaos engineering experiments, hypotheses are tested by injecting turbulence e.g. faults, under real situations, while observing system behavior. The observed behavior is *new knowledge*, as it affords insights to how the system will fail or withstand (confirm or disprove the defined hypotheses). However, the state-of-the-art chaos engineering techniques focus on availability experiments, where hypotheses are framed around availability attributes e.g. latency. We believe that security-focused hypothesis are also possible, and would be very beneficial to security professionals. Furthermore, resiliency is not only crucial to availability, but also security(confidentiality and integrity). Therefore, the next subsections lay the foundation for making these connections.

C. DEPENDABILITY

Dependability is a global concept that includes several attributes: reliability, availability, integrity, availability, maintainability and safety [26], [27]. These attributes, also illustrated in Figure 2, are highly desirable, but could be negatively impacted by the effects of *failures*, *faults* and *errors*. *Chaos engineering* implementations e.g. Netflix Chaos Monkey [8] employ *fault injection techniques* for addressing these threats. Consequent upon the success of the Netflix Simian Army, several implementation of the principles of chaos engineering have emerged. These implementations, address resiliency issues at various abstraction layer, as shown on Table 1. However, over 95% of these tools focus on

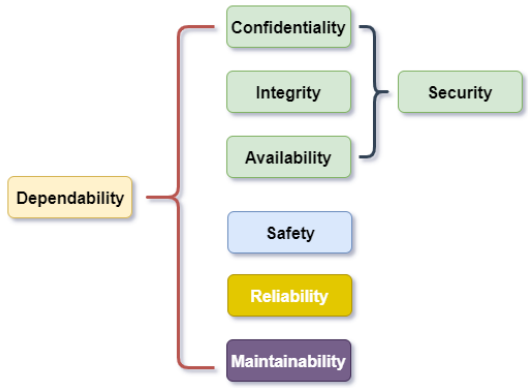


FIGURE 2. The dependability tree [26] shows the relationship between dependability and security.

availability challenges. Consequently, the remaining dependability attributes (illustrated in Figure 2) are neither tested nor guaranteed. Hence, security failures such as those caused by malicious faults (cyber attacks) are currently not handled by the current resiliency techniques: *timeouts, retries, and fallbacks*.

D. SECURITY

Security is a summation of confidentiality, integrity and availability, and is also categorized under dependability [26], [28]. These attributes define the way security of any system is perceived, hence the violation of these attributes indicate compromise of security. Traditionally, security controls are enforced to detect these security violations. However, the cloud operating model differs from on-premises data- centers, where the afore-mentioned security controls were designed to operate. Consequently, these traditional security controls are largely ineffective in cloud infrastructure. We propose the use of SCE as a proactive measure to overcome the contemporary cloud security challenges. Therefore, CloudStrike shifts focus from focusing on injection of non-malicious faults, to malicious faults. This enables verification of cloud security properties, e.g. configurations of AWS S3 buckets. A typical example is provided in the running example (Section II-A), where the attacker was able to escalate privileges (**EP02**) and move laterally without triggering any security alerts (**EP03**). Several cloud security best practices have been proposed e.g. *the principle of least privilege*. Yet, there are no defined techniques for verifying correct implementation, hence the high rate of cloud breaches. CloudStrike is designed to breach this gap via automatic and continuous verification of cloud security properties, these properties are defined as *hypotheses* for chaos engineering experiments (Section III-D).

E. RESILIENCY

Resiliency is defined as the ability of a system to persist its dependability over a period of time regardless of changes [29], [30]. These changes are very important in the cloud due to the heterogeneity of services, dynamic events and high volatility of resources. Essentially, efficient change control is imperative for cloud security as changes could be *Indicators of Compromise* [1]. Therefore, mechanisms that are designed to check for the resiliency of cloud systems should inject both malicious and non-malicious changes as part of resiliency testing. This approach efficiently tackles the cloud threat: *lack of efficient change control mechanisms* as outlined in the CSA top cloud threats 2019 [1].

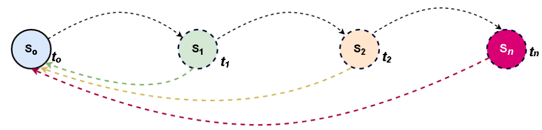


FIGURE 3. State transition analysis leveraged to enable safety in RDFI using reversibility concepts.

Algorithm 1 Disable Logging in AWS S3

```
1: procedure Disable Logging
2:     getCloudBuckets()   ▷ enumerate the buckets in the cloud
3:     selectRandomBucket ← getCloudBuckets()  ▷ select a random bucket from the set of enumerated buckets
4:     disableBucketLogging()  ▷ stop all logging activities against the bucket
5: end procedure
```

F. SAFETY IN FAULT INJECTION

Practicing chaos engineering in production requires a good measure of safety. These safety measures provide options for rolling back changes that adversely impact deployments. We leveraged the concept of *state transition analysis* to achieve safety. State transition analysis is an analytical model for detecting and representing malicious events in computer systems [31]. Essentially, malicious activities are modeled as the transition of states originating from a secure state (good) S_o. As illustrated in Figure 3, the states change from S_o to S_1 and can progress until S_n. Each subsequent state represents a compromised state due to malicious attacker action e.g. change of an AWS access policies order to escalate privileges. Therefore, the secure (good) state S_o, has to be initially established, this is straightforward if Infrastructure as Code (IaC) e.g. HashiCorp Terraform[2] or AWS CloudFormation,[3] is the orchestration strategy for the cloud environment. IaC enables declarative, representation of infrastructure in JSON or YAML formats. Conversely, IaC can be persisted and retrieved to recreate resources by rolling back changes from S_n to S_o. Note that S_o can also be constructed in the absence of IaC by enumerating and persisting cloud resources using cloud APIs.

III. RISK DRIVEN FAULT INJECTION

RDFI implements chaos engineering from a security risk-driven perspective. The security attributes (confidentiality,

[2] https://www.terraform.io/
[3] https://aws.amazon.com/cloudformation/

integrity and availability) are considered while exploring the *fault space* i.e. the hypothesis are framed within this context. Therefore, faults that impact on these attributes are orchestrated against the target cloud infrastructure. A security risk-driven approach is more helpful to security practitioners since detected vulnerabilities are analyzed and quantified, thus enabling subsequent decision-making easier and straightforward. In following subsections, several aspects of RDFI are discussed including the Execute Monitor Analyze Plan over-a-shared Knowledge-base (EMAP-K) Feedback Loop, security risk metrics, fault models and implementation of chaos engineering in RDFI.

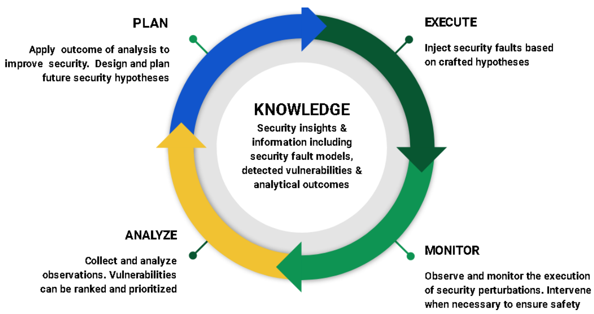


FIGURE 4. RDFI feedback loop - an adaptation of the MAPE-K framework to support security fault injection campaigns and transfer of information to security mechanisms.

A. RDFI FEEDBACK LOOP

There are currently no established guidelines for practicing SCE. However, such practices exist for chaos engineering, infact, modern software engineering frameworks e.g. microservices implement resiliency patterns e.g. timeouts and bulkheads and circuit-breaker. The RDFI Feedback Loop shown in Figure 4 summarizes how SCE can be used to ensure security and resiliency in cloud infrastructure. It describes the strategy for conveying the security information gained from the chaos engineering campaigns to the deployed security controls and mechanism in an efficient and iterative manner. The idea for adopting a feedback loop is motivated by *control engineering* and *autonomous computing* domains where the MAPE-K feedback loop [20] is a popular mechanism maintaining stability. However, the MAPE-K feedback loop is passive since it listens to events i.e. employs event-driven approaches. Conversely, SCE initiates events via fault injection and then monitors, hence a proactive feedback loop is more suitable. Therefore, we have adapted the MAPE-K feedback loop by making the *EXECUTE* phase the first module, *aka* EMAP-K. The mapping of the various MAPE-K functions with CloudStrike is shown in Figure 5. Our adapted model works as follows:

1) EXECUTE

The first component of the RDFI Feedback Loop is the *execute* component. It is responsible for injecting security faults into the target cloud infrastructure. For example, in Algorithm 2, the faults injected disable the logging

---

Algorithm 2 Malicious User-Bucket Attack Scenario

procedure BucketAttack
    createNewUser()   ▷ create a new user e.g. Bob
    getCloudBuckets()   ▷ get a list of all the buckets in the cloud
    selectRandomBucket ← *getCloudBuckets*()   ▷ select a random bucket from the set of buckets in the cloud
    createBucketPolicy()
    assignUserAccessPolicy ← *selectRandomBucket*   ▷ give user e.g. Bob read access to the existing bucket
end procedure

---

functionality of a specific AWS S3 bucket. This evasion technique is commonly used by attackers to hide their tracks and avoid detection. The execute component is responsible for implementing these fault injection operations e.g. Algorithms 2 and 2. Unlike the MAPE-K model, where the *monitor* component is the initiating component, the *execute* component initiates the EMAP-K model. This is because chaos engineering is a proactive mechanism and not a reactive one like MAPE-K.

2) MONITOR

Following the successful injection of security faults, it is critical to maintain real-time visibility of the target cloud infrastructure. This enables timely intervention if the impact of fault injection campaigns begins to adversely affect the system, especially in production environments. Therefore, the *Monitor* components uses several mechanisms to ensure visibility. Firstly, logs from CloudStrike are collected and analyzed, and thereafter observability tools from the cloud service are leveraged e.g. AWS CloudWatch [32], AWS X-RAY(distributed tracing) [33].

3) ANALYZE

Observations derived from the cloud infrastructures is collected and analyzed. The analysis helps in refining the information to aid better understanding and identification of implications e.g. impact of the security risks. Furthermore, detected vulnerabilities are scoring using security risk metrics, more details of the scoring methodology is provided in Section III-B.

4) PLAN

The *Plan* component takes the security information acquired during the fault injection and applies it in two major ways. Firstly, the security information, is passed to the respective security mechanisms e.g. security tools deployed to protect cloud infrastructure where it is used for possible hardening measures. Hence, the security controls, as enumerated on Table 2 leverages the security information to implement several security measures: preventive, detective predictive and recovery. For example, security fault injection campaigns orchestrated against the AWS infrastructure in *Running Example* would identify over-privileged IAM policies

TABLE 2. Dependability VS security controls.

Means of dependability	Security controls
Fault prevention	IDS, firewalls, AWS GuardDuty
Fault tolerance	Intrusion Prevention Systems
Fault removal	Vulnerability Patching, access privilege revocation
Fault forecasting	Threat/vulnerability prediction, threat intelligence analytics

and flag it as a security control violation. Consequently, a less permissive policy could be proposed to replace the existing one. The second way the security information is applied consist in preparing for subsequent fault injection campaigns. The discovered vulnerabilities are leveraged to plan more attacks for other assets in the cloud infrastructure e.g. by enriching the fault models.

5) KNOWLEDGE-BASE

At the center of the RDFI feedback loop is the knowledge-base, consisting of security information. The security information about the cloud infrastructure is derived from several sources e.g. fault models and cloud security best practices and secure baselines. Also, the results of the analyzed behavior due to fault injections is an important part of this knowledge-base as it provides information that is immediately actionable. For example, if there are no detected vulnerabilities due to the security faults defined in Algorithm 2, that observation is persisted in the knowledge-base. A different fault will be injected into that specific cloud resource in the future fault injection campaigns.

B. SECURITY RISK METRICS

The outcome of fault injection campaigns are not left in binary categories e.g. *secure/insecure* or *true/false*, instead fine grained security risk metrics are employed. These metrics are computed for every security vulnerability detected during fault injection campaigns using the Common Vulnerability Scoring System (CVSS), one of the most popular security metrics standard.

1) CVSS

We extended our previous works on threat modeling and proactive risk analysis for cloud infrastructure, where we used the CVSS version 2 to score vulnerabilities in cloud infrastructure [34], [35]. The CVSS metrics are expressed using with *base scores*, which are numeric representations of risks, assessed in terms of severity [36], [37]. The *base scores* are computed using the *Impact* (Eqn 2) and *Exploitability* (Eqn 3) metrics, as expressed in Eqn 1. We have used our expert knowledge to compute these metrics, comparing them with similar vulnerabilities and following the guidelines in the CVSS manuals [36], [37]. Therefore, detected vulnerability due to the fault injection campaigns are scored, and the scores serve as a guide for risk prioritization [38] and other risk management tasks, thereby making our approach more practically useful.

2) DERIVING SECURITY METRICS WITH CVSS

Let us consider how to compute security severity using the CVSS for two representative cloud attacks: *Cloud Storage Enumeration Attack* and *Cloud Storage Exploitation Attack* [34], [35], [39].

- *Cloud Storage Enumeration Attack*. This attack aims at detecting misconfigured buckets for a selected target e.g. a company's AWS S3 buckets that are publicly accessible. The attacker leverages previous knowledge about the target acquired via enumeration techniques [40], to construct possible keywords that are relevant to the target e.g company name. These keywords are then fed into the word-list generation tool e.g. *Mentalist*,[4] to generate all possible word combinations that are potentially AWS S3 bucket names. Thereafter, the generated word-list is fed to a cloud exploitation tools e.g. Bucketfinder[5] to conduct the attack. Bucketfinder uses the word-list to construct and probe AWS S3 URLs using HTTP GET requests, responses with code 200 are publicly accessible. Due to space limitations, some details of the Equations 1 - 2 are omitted e.g. static values for the AccessVector, AccessComplexity, Authentication, ConfImpact, IntegImpact and AvailImpact. These values are available at various resources e.g the CVSS Implementation Guide [36]. We assign *Network* for the AttackVector metric since the attack can be executed over the internet. the AccessComplexity is assigned *Low* given that attackers can execute this attack with tools available *in the wild* e.g Metasploit and on several GitHub repositories. The Authentication metric is set to *None*, because no authentication is required for the attack. For the Impact metrics, IntegImpact, ConfImpact and AvailImpact is set to *Partial* since there is a possibility of either acquiring materials encrypted in buckets/objects with properly configured Access Control List (ACL). Based on these metrics *(AV:N/AC:L/Au:N/C:P/I:P/A:P)*[6] we derive 7.5, as the base score. The Cloud Storage Enumeration

[4] https://github.com/sc0tfree/mentalist

[5] https://digi.ninja/projects/bucket finder.php

[6] this is a vector string representation of all computed metrics for a vulnerability

$$BaseScore = round\_to\_1\_decimal(((0.6 * Impact) + (0.4 * Exploitability) - -1.5) * f(Impact)) \quad (1)$$

$$Impact = 10.41 * (1 - (1 - ConfImpact) * (1 - IntegImpact) * (1 - AvailImpact)) \quad (2)$$

$$Exploitability = 20 * AccessVector * AccessComplexity * Authentication \quad (3)$$

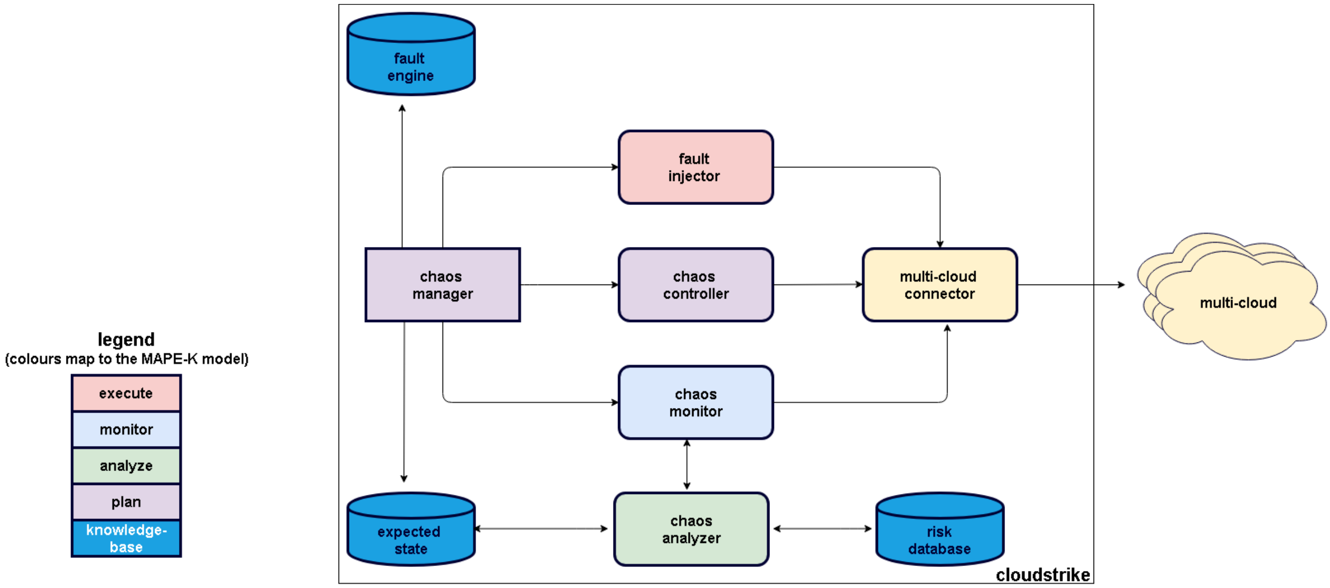


FIGURE 5. Architecture of CloudStrike showing the mapping to MAPE-K model.

Attack is comparable to *brute force password guessing attacks* e.g. CVE-2012-3137.[7]

- *Cloud Storage Exploitation Attack* The *Cloud Storage Enumeration Attack* could use the previous attack as a staging step. The actual attack against identified misconfigured buckets are during this attack, using cloud exploitation tools e.g. Bucketfinder. To compute the severity scores, we assign *Network* to the AttackVector metric, since the buckets are reachable via the internet. The AccessComplexity is assigned *Low*, while the Authentication metric is set to *None*, given there is no authentication mechanism protecting the bucket. The Impact metrics is more severe given the previous attack informs the attacker of buckets that are *publicly accessible*. Thus, the IntegImpact, ConfImpact and AvailImpact are set to *Complete*. We thus have the base metrics as *(AV:N/AC:L/Au:N/C:C/I:C/A:C)*, and arrive at a score of 10.0. The score is reasonable considering it affords an attacker full access to AWS S3 bucket.

C. FAULT MODELS

Fault models [41] are commonly used in traditional fault injection schemes to establish a sequence and order for conducting fault injection campaigns. In order to derive the fault models used in our scheme, several sources of security information have been synthesized. Furthermore, an important aspect of fault injection algorithms is the ability to detect all possible faults (wide *fault coverage*) within a defined failure scope. Essentially, our failure scope encapsulates the impact of security failures against cloud assets. We based our fault models on the CSA cloud penetration test playbook [23], which categorizes public IaaS into three domains (see Figure 8) for security testing: (1) *application, data, business logic*, (2) *cloud service* and (3) *cloud account*. However, we focus on the latter two domains: cloud account security and cloud service security which directly map to the cloud IAM and cloud storage respectively. The following sources considered to formulate RDFI fault models:

1) CLOUD SECURITY KNOWLEDGE-BASE

Cloud security best practices have been proposed by several organizations such as the CIS benchmarks and the CSA security guides. These best practices specify *checks* to improve the security posture for CSPs and cloud customers. Automated security tests could therefore be implemented based on these best practices. For example, the AWS CIS Recommendation 2.6 recommends activation of AWS CloudTrail: *Ensure S3 bucket access logging is enabled on the CloudTrail S3 bucket*. This recommendation aims at ensuring that all activities against the AWS buckets are recorded and retained for subsequent retrieval and analysis. Accordingly, an example of a security fault injection we have derived from this recommendation is *disable_bucket_logging*. Algorithm 1 illustrates the implementation of this fault against AWS S3 buckets.

2) CLOUD PENETRATION TESTING PLAYBOOK

Although the above-mentioned approach provides rich guidelines for building *fault models*, we leverage existing knowledge from traditional security testing e.g. penetration testing. This is achieved by synthesizing the *test cases* provided in the CSA Penetration Testing playbook [42], which contains over 70 test cases. A key advantage of the playbook is that it puts the test cases within the context of public clouds and extracts the responsibilities that are specific to the Cloud Customer (CC). The test cases are generic and therefore applicable to different cloud platforms. There are several categories of

[7] https://nvd.nist.gov/vuln/detail/CVE-2012-3137

TABLE 3. Security fault injection categories - drawn from the CSA penetration testing playbook [23].

Test ID	Test category	Examples
T01	Validating baseline security requirements	Change enterprise IAM permission e.g. use HR credentials to access development environment
T02	Employ security test cases, guides and checklists relevant to domain & technologies	cloud applications
T03	Test for Spoofing of user identity and other entities	Compromise default privileged service and user accounts in legacy cloud environments and services
T04	Test for Tampering	Alter data in data-store for fraudulent transactions or static website compromise (AWS S3, RDS, RedShift)
T05	Test for Repudiation	Operate in regions where logging is not enabled or disable global logging (like CloudTrail)
T06	Test for Information disclosure e.g. privacy breach or data leak	Leverage misconfigured and default security groups and access lists for ex-filtration of data to any internet IP address (VPC ACL, instance security groups)
T07	Test for Denial of service	Modify cloud services configuration, data-stores and/or accounts
T08	Test for Elevation of privilege	Add users, assets or accounts to existing roles or groups with higher privileges leverage privileges such as iam:AddUserToGroup
T09	Test for Other Cases and Objectives	Leverage misconfigured security groups and ACLs for lateral movement between assets in the Cloud (EC2, RDS, other), from account to account (AWS cross-account assume role)
T10	Persistence	Assign a public IP to a compromised / internal resource (AWS CLI / console - elastic IP)

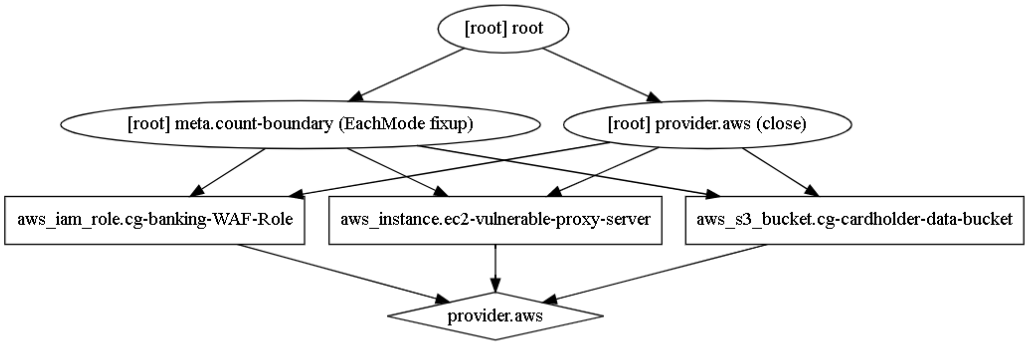


FIGURE 6. An attack graph of the running example (Note: portions of this graph were omitted for legibility).

security tests can be performed, some of these are shown on Table 3.

3) ATTACK GRAPHS

One limitation of the above fault models is the lack of methodologies for sequential injection of faults. In reality, attacks are conducted in a step-by-step procedure i.e. from unprivileged to privileged states to achieve desired objectives. Furthermore, the chaining of attacks is a common attack technique employed to hide malicious tracks or persist control e.g. *cyber-attack kill chain* [43] is a popular attack model that defines methods of advanced persistent attacks. Therefore, RDFI employs *attack graphs* [44], which are commonly used to illustrate such steps employed by attackers. This approach is similar to Lineage Driven Fault Injection (LDFI) [45], in which a top-down approach is used to inject faults into a system to observe the success rate of the system (consequences). Attack graphs are also similar to *fault trees*, which are commonly used to illustrate fault models. Furthermore, attack graphs aid in avoiding randomized attack procedures as practiced in other chaos engineering tools e.g. Chaos Monkey [45]. Another advantage of using attack graphs is they aid automation, and reduce the need for security experts and chaos engineering experts as noted by Alvaro et.al [46]. We leverage the graph generation feature of Terraform[8] to construct attack graphs (Figure 6), which are then further processed using GraphViz-Java,[9] a Java implementation of the of GraphViz. This is quite straightforward since Terraform internally depends on *Resource Graphs*, to perform its operations e.g. *terraform apply*. Furthermore, this feature internally uses GraphViz [10] and Dot,[11] which are popularly used for graph visualization and expression language respectively. Attack graphs can also be constructed for cloud infrastructure orchestrated using other tools by discovering the infrastructure Terraform *resource discovery* feature.[12] In this case the cloud infrastructure is first converted to Terraform state files to enable graph generation.

D. APPLYING CHAOS ENGINEERING WITH RDFI

CloudStrike uses several chaos algorithms to inject security faults (*AttackPoints*) into cloud infrastructure, thereby

[8] https://www.terraform.io/
[9] https://github.com/nidi3/graphviz-java
[10] https://www.graphviz.org/
[11] https://graphviz.gitlab.io/_pages/doc/info/lang.html
[12] https://www.terraform.io/docs/providers/oci/guides/resource_discovery.html

TABLE 4. Examples of CloudStrike's AttackPoints.

Attack ID	Cloud Resource	Chaos Action	Description
AP1	User	create	create random user
AP2	User	delete	delete existing user
AP3	User	modify	change user configuration e.g. privileges, role or group
AP4	Policy	create	create new policies with random ACLs and attach to cloud resource(s)
AP5	Policy	modify	modify existing policy e.g. change ACL to deny original owner access to the resource
AP6	Policy	delete	detach policy from a resource, delete the policy
AP7	Role	create	creae a new role
AP8	Bucket	make public	alter private configuration to public
AP9	Bucket	disable logging	stop logging API calls against bucket
AP10	Bucket	make unavailable	simulate bucket unavailability e.g. by changing bucket ACL from ALLOW to DENY

causing specific actions. Table 4 outlines some of these attack points and the specific cloud resources that are impacted. In general, the chaos engineering principles proposed by Basiri et.al [16] are adhered to as explained below:

1) BUILD A HYPOTHESIS AROUND A STEADY-STATE BEHAVIOR

Central to every chaos engineering experiment is the determination of a hypothesis about *normalcy* and *abnormality*, with corresponding measurable attributes. Thus, we exploited the concept of *expected state* [34] - the secure state of a cloud resource at time t_o. Essentially, the expected state is known by the resource orchestration system. For example, an ACP may specify *read* access for a user, *Alice* at time t_o. This is registered in the orchestration system and a measurable attribute is defined e.g. a HTTP 401 error (*unauthorized*) is produced if Alice sends a request to a resource (e.g. bucket) after her privileges are removed.

2) VARY REAL WORLD EVENTS

To simulate real world events, a variation of possible attacks is implemented. CloudStrike orchestrates random actions against target cloud systems e.g. deletion, creation, and modification, using the respective cloud APIs. Three chaos modes are supported: *LOW*, *MEDIUM* and *HIGH*, which correspond the magnitudes of 30 %, 60 % and 90 % respectively. Table 4 is an example of *AttackPoints* used, each AttackPoint defines a specific action to be conducted, a combination of two or more AttackPoints forms an *attack scenario*. Algorithm 2 combines $AP1$ and $AP4$ to create a scenario where an attacker creates a random user in a cloud account, creates a privileged policy for accessing a cloud bucket and attaches the policy to the malicious account.

3) RUN EXPERIMENTS IN PRODUCTION

Chaos engineering experiments take a different approach from traditional software engineering testing, where tests are limited to development environments [17]. Since the major motivation for Chaos engineering is to gain confidence when systems are exposed to *real-life scenarios* i.e. production, running experiments in such environments is imperative. However, a phased approach is required based on the level of maturity of the organization. These levels of maturity are clearly outlined in the chaos maturity model [17] and are hinged on two core metrics: *sophistication and adoption*. Safety measures are required as a fundamental basis for recovering systems to *steady states*. We achieve this by employing the concept of *expected states* and *cloud state* [34]. These expected states are persisted and can be easily used to recover cloud environments to its secure states. We deployed CloudStrike against resources deployed on AWS and GCP.

4) AUTOMATE EXPERIMENTS TO RUN CONTINUOUSLY

A clear distinction between traditional security testing and chaos engineering is the use of automation. Security automation enables continuous oversight, which is necessary in the cloud due to constant changes e.g. change of ACPs and provisioning of new API keys. These changes could be initiated for either malicious or benign reasons hence the need for proactively measures to experiment and study malicious scenarios, thereby gaining insights into efficient ways for designing and implementing secure cloud systems.

5) MINIMIZE THE BLAST RADIUS

The *blast radius* refers to the extent to which the impact of a fault injection campaign might extend, in terms of severity and reach. Reducing the blast radius is an important step to control the risk of system failure, especially against production systems. At the Netflix Chaos Team, common techniques implemented to deuce the risks included sampling of requests, and use of sticky sessions [24]. Here we leverage the already discussed safety techniques in Section II-F, which employs the ability to rollback *good states* based on the principle of *state transition analysis*.

IV. IMPLEMENTATION

All components of *CloudStrike* are implemented in Java, attacks are transmitted to the cloud platforms using APIs of AWS and GCP, hence there is no need to install agents on target cloud infrastructure. Figure 5 illustrates CloudStrike's architecture, details are as follows:

A. CHAOS CONTROLLER

This is *the coordinator* of the chaos injection experiments, it receives requests for experiments with necessary parameters e.g. cloud access credentials, preferred chaos mode and cloud resources to be tested. This is based on a designated security hypotheses and it is passed down to the Chaos Manager. Eventually, the results of the chaos engineering experiments e.g. the detected vulnerabilities are analyzed and handed back to the chaos controller for onward transmission to human administrators or external security tools. The Chaos controller maps to the *plan* component of MAPE-K framework (Section III-A4).

B. CHAOS MANAGER

The Chaos Manager receives the instruction to conduct attacks based on specified *attack modes*. The attack modes are categorized as follows: *LOW, MEDIUM & HIGH*. However, to have more refined, fine-grained control, the *rate of attack*, which is abstracted in the aforementioned attack modes, could be varied from 0.1 - 0.9, where 0.9 refers to more aggressive attacks. Thereafter, the Chaos Manager aggregates the specified targets from the *expected state* (see Figure 5), then a subset of the collected set of assets is selected based on the attack rate. The higher the attack rate, the more the number of assets to be attacked. The selected assets are them attacked based on *RULES* drawn from the Fault Engine e.g. *DELETE* AWS S3 bucket *X*. The Chaos Manager maps to the *plan* component of the MAPE-K framework (Section III-A4).

C. FAULT ENGINE

The fault engine maps to the *knowledge-base* component of the MAPE-K framework (Section III-A5). It consists of aggregated knowledge on about cloud compliance, best practices and attack graphs as described Section III-C. These information is thereafter translated into actionable code, in the form of *RULES* that define specific *ACTIONS* against specific *ASSETS*. Here, we define an *ACTION* as *what has to be done against an asset*, these could be:*create, delete, modify*, which will create, delete and modify the cloud resource respectively. Similarly, the *ASSETS* refers to the cloud resource involved, e.g. AWS S3 bucket or AWS IAM policy. For example, in the running example detailed in Section II-A, a *RULE* will be of the form: *MODIFY ACL for BUCKET X TO DENY ACCESS TO USER Y*, in this case the chaos algorithm will fetch the ACL for Bucket X and remove User X name from it. Effectively, User X will no longer have access to the Bucket.

D. FAULT INJECTOR

The fault injector is responsible for implementing the security faults composed by the Chaos Manager. The faults are orchestrated against the target cloud assets. Furthermore, using a defined heuristic, the fault injector either injects single attack points or combines multiple attack points into attack scenarios as illustrated on Table 4. The Fault Injector maps to the *execute* component of the MAPE-K framework (Section III-A1).

E. CHAOS MONITOR

To ensure safety, the *chaos monitor* continuously monitors the progress of attacks to easily detect overwhelming effects due to fault injection. *We employ techniques that afford reversibility of states as described in* Section II-F. We leverage our previously developed system CSBAuditor [34], for maintaining continuous visibility of monitored cloud accounts. This is supported by a logging system based on Log4J,[13] and Cloud provider logging mechanisms: AWS CloudWatch and GCP Stackdriver. Combining both server-side and cloud-side logging and metrics provides efficient observability for deriving real-time insights of chaos engineering experiments. The Chaos Monitor is also responsible for recovering the target system to normal (secure) states either when the experiment is terminated or completed (Figure 3). The chaos monitor implements the *state transition analysis* to reverse the effects of experiments. It also computes the risk scores using the CVSS algorithms in Eqn 1 - 3 and persists the reports generated in the risk database. The Chaos Monitor maps to the *monitor* component of the MAPE-K framework (Section III-A2).

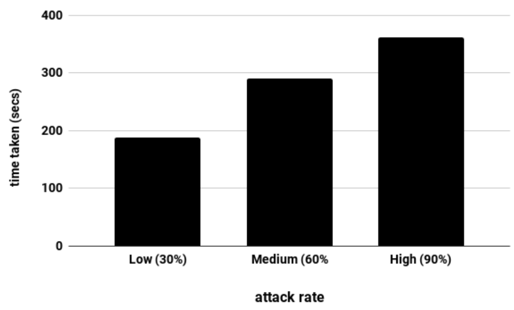
F. CHAOS ANALYZER

The vulnerabilities detected during fault injection campaigns are passed to the Chaos analyzer for subsequent analysis. Here, pre-computed severity scores are assigned to the vulnerabilities and reports are generated. Furthermore, the observations are retained in a knowledge-base (risk database) for later reference and also used for implementing security counter-measures and mitigation. Possible recommendations include updating security rules for security groups (cloud firewalls), restriction of access to overly permissive access control policies. The results of the analysis are also passed to the Chaos Controller for onward transfer to the relevant security mechanisms to remediate the detected vulnerability. Therefore, the Chaos Analyzer maps to the *analyze* component of the MAPE-K framework, while the risk database maps to the *knowledge-base* component (Section III-A5).

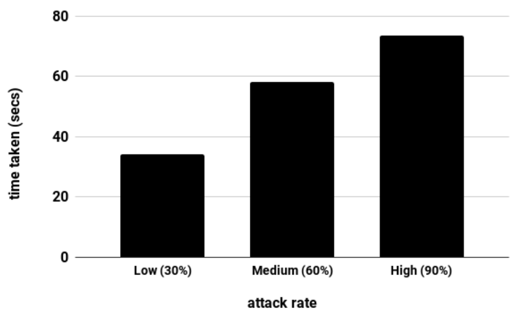
V. EXPERIMENTS AND EVALUATION

We evaluated CloudStrike against a cloud infrastructure testbed that depicts an enterprise cloud environment, comprised of assets deployed on AWS and GCP. We adopted the cloud testing methodology proposed by the CSA's Cloud Penetration Testing Playbook, which groups cloud infrastructure into three security domains: (1) Application Data, Business logic, (2) Service and (3) Account. Figure 8 clearly illustrates these domains with more details.

[13] https://logging.apache.org/log4j/2.x/



(a) GCP Performance



(b) AWS Performance

FIGURE 7. Comparing (a) GCP and (b) AWS performance.

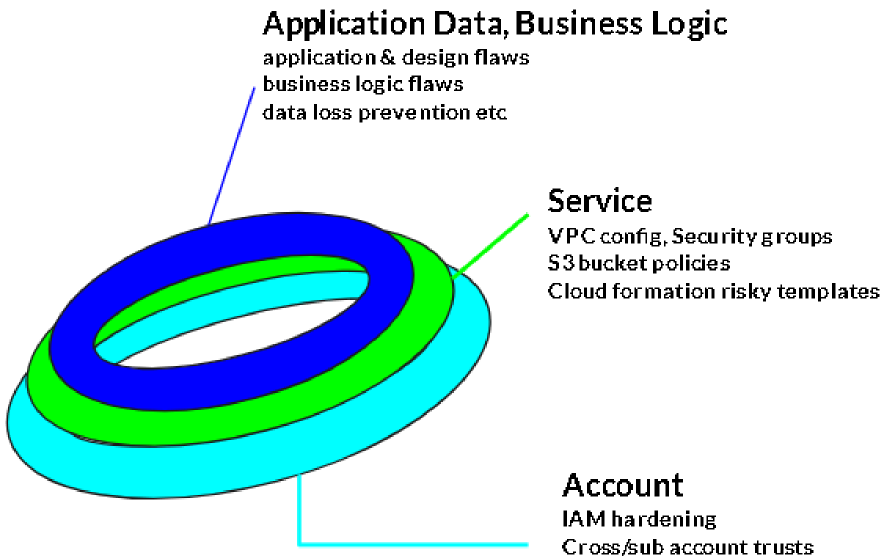


FIGURE 8. Three-layered cloud infrastructure for security testing.

- Cloud Test-bed: Our experiments are focused on IAM (users, policies e.t.c.) and cloud storage service (S3 buckets, configurations e.t.c.), which are in categories (2) and (3). We do not consider the third component i.e. the application layer. Fifty user accounts are provisioned on AWS and GCP cloud infrastructure, 25 users per cloud. Each user account is properly configured using privilege separation concepts.
- CloudStrike deployment: CloudStrike is deployed on a Windows 10 computer, composed as follows: Intel (R) Core (TM) i5-5200U CPU, 2.20Ghz processor, 8GB RAM and 1 TB HDD.

A. TIME PERFORMANCE

These set of experiments aim at evaluating the performance of CloudStrike *w.r.t* time overhead while injecting security faults (*workloads*). For the first experiment, the major attack modes *LOW, MEDIUM and HIGH* produced by the Chaos Manager (Section IV-B), are launched against GCP assets. After each attack mode, the assets are recovered back to the secure state using the *expected state* [34] - the secure state of a cloud resource at time t_o. Details of our recovery strategy is in Section II-F. Essentially, the expected-state is the *single-source-of-truth*, hence is used to recover the test-bed to its expected-state. The Chaos Manager is used to construct and similar faults against the AWS assets, the time taken for each step is recorded. Figures 7a and 7b show the results for GCP and AWS respectively. We note that the performance for AWS is better than GCP, for the *LOW* attack modes, it takes about 290 secs to complete the attack for GCP. Conversely, the same attack mode (*LOW*) is completed within 38 seconds for AWS. Similar disparities in time performance is observed for other experiments, essentially the GCP APIs are more complex, having layered dependencies and more calls are made to complete requests. The next experiment is similar to the previous ones, but only one attack rule is used: *public_bucket_access*. This rule is used for making private buckets public, hence the *expected-state* is first enumerated, to acquire the details of the buckets and respective ACLs. A subset of random buckets is extracted from the set of all buckets, then the ACLs of the randomly selected subset of buckets are changed from *PRIVATE* to *PUBLIC*. Figure 9 illustrated the combined time taken to based on varying *attack-rates*. The graph is plotted on a scale of 0.1 to 0.9 illustrating the implemented attack rates: 0.9 depicts the most severe attacks, resulting from a higher number of compromised assets. We note that the time taken is almost linear, reflecting a linear increase of time relative to increase in attack rates. Hence, the time taken has no significant overhead to the system, implying that the system can be easily scaled (e.g. to test hundreds of cloud resources on multiple cloud infrastructure) without risking the consequence high overhead or performance challenges.

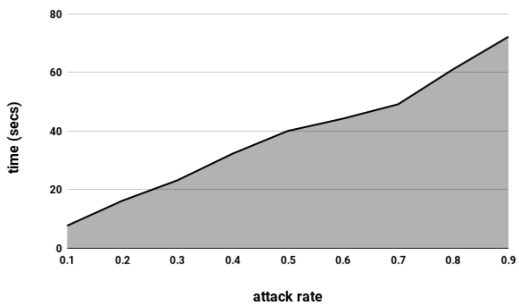


FIGURE 9. Time taken for *public_bucket_access* attacks against AWS and GCP.

FIGURE 10. Performance of CloudStrike fault injection over three modes: LOW, MEDIUM and HIGH.

B. PERFORMANCE OF RECOVERY OPERATIONS

Safety is crucial for security fault injections campaigns, as earlier explained in Section II-F. Therefore, CloudStrike performs recovery operations on completion of security fault injection campaigns. We want to evaluate the impact of these operations to gain insights of the overhead. Therefore faults are injected against the AWS test-bed using the three fault modes i.e. LOW, MEDIUM and HIGH for about 10 minutes. The results are resented in Figure 10, it shows the time on the x-axis and the number of requests executed by CloudStrike on the y-axis. Figure 11, combines the number of requests for the fault injection (in blue), and the number of requests for the recovery operations (in red). Clearly, the recovery operations require more API calls, compared to the fault injection requests. The reason for this is that the recovery operations execute global checks for all assets using the *state transition analysis* technique and *expected-state* earlier explained in Section II-F. On the one hand, this approach has the advantage of exhaustively inspecting the entire set of resources to detect and reverse changes. On the other hand, this results in a lot of HTTP requests depending on the volume of injections (mode of injection) that composed the fault injection campaign.

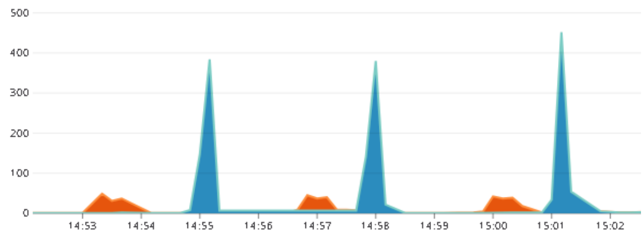


FIGURE 11. Performance of recovery operations to recover the secure baseline.

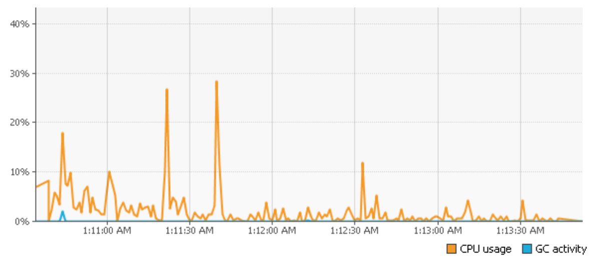


FIGURE 12. CPU performance.

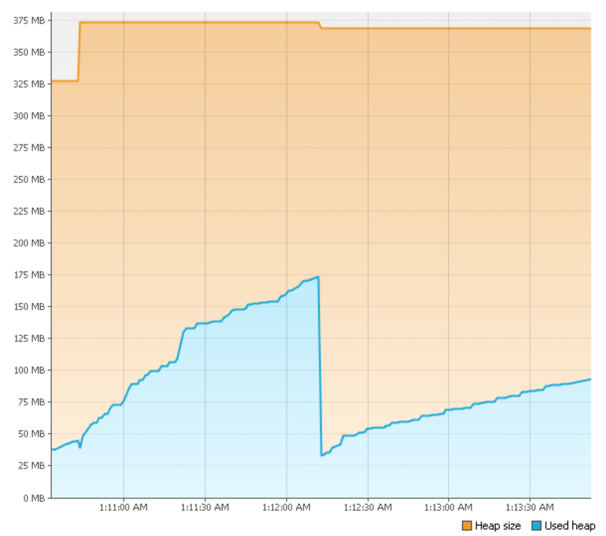


FIGURE 13. Memory consumption.

C. CPU AND MEMORY CONSUMPTION

The aim of this experiment is to analyze the overhead with regards to CPU and RAM. Figure 12 illustrates the CPU consumption of CloudStrike, we observe that on the average, the CPU consumption is less than 10 %, however, there are some spikes observed which correspond to the period when the GCP assets are attacked. The GCP API has a higher overhead due to re-authentication and more complex HTTP requests/responses, hence more CPU is utilized. Similarly, for memory consumption, we observe that memory consumption gradually rises from a minimum of 28mb to a maximum

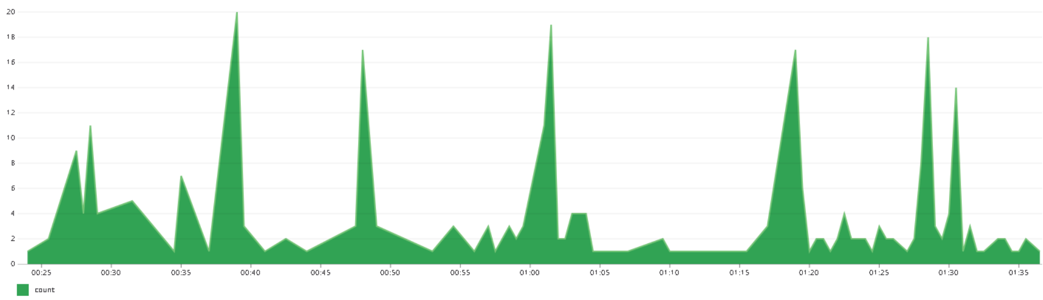


FIGURE 14. Improvement of attack detection due to application of knowledge provided by CloudStrike.

of 175mb, this corresponds also to the increased rate of attacks in two different attack cycles (Figure 13).

D. SECURITY EVALUATION

A goal of SCE is to prove security hypotheses, and thereafter, provide guidance on appropriate security countermeasures. The adapted MAPE-K model (EMAP-K) illustrated in Figure 4, provides a model for automating this process thereby making it agile and continuous. The experiments described above implemented several security hypotheses composed of attack points and attack scenarios (Figure 4). In order to evaluate the performance of CloudStrike from a security perspective, consider the attack illustrated in Algorithm 2. Here the hypothesis is *Security Alarms will be triggered if the bucket logging feature is disabled*. We chose the *bucket logging* feature since it is a best practice recommended by the CSA: *Ensure S3 bucket access logging is enabled on the CloudTrail S3 bucket* and also recommended as a *Test for Repudiation* in the CSA Penetration Testing Playbook [23]: *Test for Repudiation - Disable data store access logging to prevent detection and response T05 on Table 3*.

We describe some of the results below. Since the cloud assets are deployed on AWS, we enabled several AWS security services to i.e. anomaly detection architecture composed: AWS Detective, AWS Config, AWS GuardDuty and AWS CloudWatch. Following, the attacks implemented in the above section, we notice the only one detection captured the hypotheses tested via the fault injection campaigns.

Specifically, the only hypothesis proven right was *Alarms will be triggered if S3 bucket policies are altered*. For example, the AWS GuardDuty alarm in Listing 1 was thrown indicating the detection of malicious event, the even twas triggered due to our fault injection campaigns. However, the other faults injected were not detected by the AWS security tools. To improve the detection efficiency and security of the system, it is necessary to fine tune the detection configurations of the AWS security services. Therefore, based on the results of the fault injections, i.e. the infrastructure that was successfully compromised, we can exploit this knowledge as a guide. This is done by implemented a detection rules on AWS CloudWatch, so we the policy in Listing 2 to detect bucket policies and bucket ACLs modification events. The rule works by aggregating all access logs using Cloudtrail

```
Amazon S3 Server Access Logging was disabled for S3 bucket
        company-turcotte-log-ddbe1033-e65e
by Attacker calling PutBucketLogging. This can lead to lack
of visibility into actions taken on the affected S3 bucket
and its objects
```

Listing. 1. Security alert from AWS guardduty indicating a BucketLogging disabled event.

```
{
    "source":[
    "aws.s3"
        ],
    "detail-type":[
    "AWS API Call via CloudTrail"
    ],
    "detail":{
        "eventSource":[
        "s3.amazonaws.com"
        ],
    "eventName":[
    "PutBucketPolicy",
    "PutBucketAcl"
            ]
        }
}
```

Listing. 2. AWS CloudWatch rule for detecting malicious events.

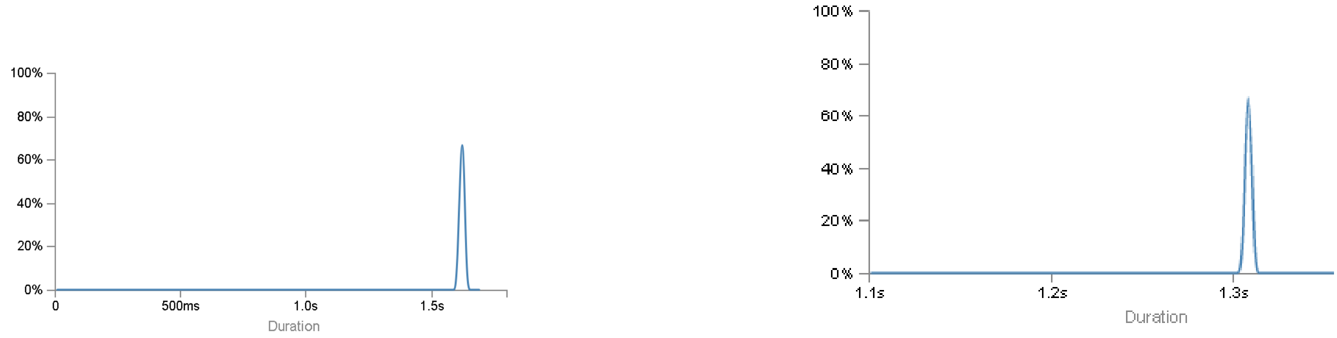
and thereafter filtering the logs for specified API calls that trigger corresponding events, we are interested in these two API calls: *PutBucketPolicy* and *PutBucketAcl*. Since Cloudtrail provides detailed history of all events, this provides an effective way for detecting when malicious requests are made. Thereafter we repeat the fault injection as above and we observe that the number of events detected by the detection system increases, this is illustrated in Figure 14.

E. COMPARISON WITH ChaoSlingr

To the best of our knowledge, ChaoSlingr[14] is the only application that provides similar functionalities with CloudStrike. The other Chaos Engineering tools either operate

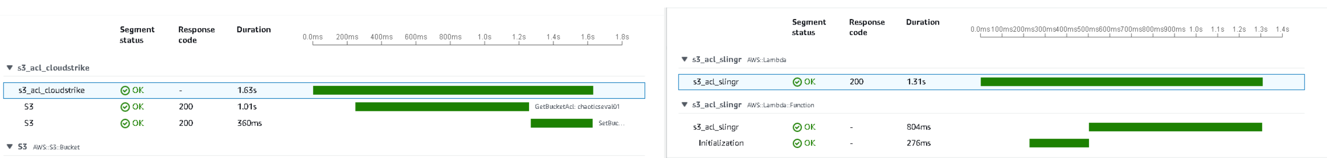
[14]https://github.com/Optum/ChaoSlingr

IEEE Access



(a) CloudStrike Performance (b) ChaosSlingr Performance

FIGURE 15. Comparing response distribution for injecting faults into AWS S3 buckets (CloudStrike VS ChaoSlingr).



(a) CloudStrike Performance (b) ChaosSlingr Performance

FIGURE 16. Comparing trace distribution for injecting faults into AWS S3 buckets (CloudStrike VS ChaoSlingr).

at a different abstraction level are designed to ensure non-security resiliency attributes e.g. availability, as summarized on Table 1. In order to compare the performance of CloudStrike with ChaoSlingr, we deployed ChaoSlingr and configured it against the environment earlier explained in the beginning of this section. ChaoSlingr was implemented as a *Proof-of-Concept* for SCE, however the tool implements only three rules: *s3_Policy_slingr, s3_acl_slingr and PortChange_Slingr*. We compared CloudStrike with ChaoSlingr using the rules that are responsible for making the S3 buckets to be public, these rules are designed to check if there are any security alerts that are triggered when the buckets are switched from *PRIVATE* to *PUBLIC*, therefore the rules *s3_acl_slingr* and *cloudstrike_acl_public* were compared respectively. A new bucket is created in the AWS environment - **chaoticseval01**, and configured to be private. Then one after the other the rules are executed against the bucket and the time performance based on time is recorded using the AWS XRAY. Figures 16a and 16b illustrate the performance of CloudStrike and ChaoSlingr respectively. As seen in the Figure 16, ChaoSlingr is slightly faster than CloudStrike with approximately 30s. However, we assume the gain in speed to be based on the the fact that ChaoSlingr is implemented as *AWS Lambda serverless functions* [47], which makes it faster since most intercommunication is between internal AWS APIs, while the communication used by CloudStrike involves more of external API since we use the basic AWS Java APIs, and CloudStrike is deployed locally deployed on a local PC. A closer look at the trace distribution as measured by AWS XRay (distributed tracing service), shows that the initialization phase for ChaoSlingr is about 276ms, while that for CloudStrike is about 360ms. Furthermore, ChaoSlingr is faster for the actual fault injection. However, CloudStrike implements over 20 fault injection rules some of these are listed on Table 4, therefore a wider fault space is covered. ChaoSlingr implements only 3 rules.

F. DISCUSSION

Currently, the aspects of most cloud security configuration involve manual efforts, this increases the chances of human error, considering the need to scale while configuring complex cloud assets like access policies [39]. There is a growing adoption of IaC and orchestration techniques, however these mechanisms are mostly not focused on security and therefore require security configuration in order to orchestrate infrastructure securely. Furthermore, security services offered by CSPs e.g. AWS CloudWatch and AWS Detective are quite immature, mostly requiring human expertise to use effectively. Hence, SCE provides a test-based approach that provides clearer guidance on which security efforts to focus and what configurations are not secure. Though some security knowledge might be required, the goal is to produce reports that are clear and direct, stating the detected vulnerabilities and recommended solution. For example, in the security evaluation (Section V-D), the detection efficiency of AWS CloudWatch improved following implementation of the rules necessary to mitigate the vulnerabilities detected by CloudStrike. However, the rules were added manually, and will not scale in reality, therefore automating the entire process will be an interesting future effort. Similarly, the use of attack graphs would aid automation and integration with other tools. We consider this a huge gain for security professionals not conversant with cloud technologies since attack graphs are well known. However, the performance of the attack graph was not evaluated in this work as we focused on the results produced using random fault injection techniques, therefore, the attack graph analysis is planned for future work. We also acknowledge that most of the evaluation focused on AWS, this is due to the concentration of tools and methodologies around AWS technologies. Furthermore, the cloud threat landscape has seen more attacks focused on AWS infrastructure such as the Capital One data breach which we used as a running example (Section II-A). However, we have developed the chaos algorithms and other components of CloudStrike to also test resources on GCP.

VI. RELATED WORK

There is a limited amount of work on resiliency testing of distributed systems using chaos engineering techniques, and most of these work aim at tackling the non-security attributes. Conversely, security fault injection has been used in contexts other than cloud systems. We compare and contract our work with these two categories of related works i.e those that focus on non-security attributes and those that investigate security attributes.

A. NON-SECURITY FAULT INJECTION

Chaos Monkey [8] is a tool invented by Netflix for injecting random faults in production. Together with its variants (Netflix Simian Army), perturbations are injected into various levels of cloud infrastructure including VMs, to cloud network regions and availability zones. Through these means, various resiliency issues are detected especially at the network levels. However, the faults injected specifically test the availability related attributes of cloud services. We aim at the security attributes in order to introduce resiliency that improves security. Moreover, Chaos Monkey injects faults in a random manner, we aim at employing sequential fault injection strategy via RDFI. Gremlin [19] is a fault injection system aimed at testing the resiliency of microservices. It achieves its objective by injecting non-malicious faults against the network layer of microservices. Our fault injection strategies leverage the API connecting cloud customers and cloud services and focus on security faults. Zhang et'al [10] proposed *ChaosMachine*, a system for live analysis and falsification of exception-handling in the JVM. ChaosMachine employs bytecode instrumentation and remote control of fine-grained fault injection to detect resilience weaknesses in *try-catch-exemption* handling. These issues are thereafter reported to developers via reports. Alvaro et'al proposed LDFI [45], as an alternative to random fault injection to provide structured and intelligent exploration of defined fault space. We gained the intuition for employing attack graph for exploring cloud infrastructure attack surfaces (which defines the fault space from a security viewpoint) since LDFI does not suit the security use-case.

B. SECURITY FAULT INJECTION

Du 'et al [48] proposed an approach for detecting vulnerabilities in software systems via injection of security faults. The fault models employed were extracted from vulnerability databases. Similarly, Fonseca et.al [49] proposed Vulnerability & Attack Injector Tool (VAIT) for automatic injection of security faults into web applications. Similar to our work, security faults are injected based on the continuous analysis the target web application and the injected attacks are realistic. In [50], a fault injection taxonomy for Service-Oriented Architecture (SOA) is proposed, the taxonomy includes security faults such as authentication and authorization faults. Infection Monkey[15] is a open source *Simulation As* tool used for validating the resilience of cloud networks and compute instances. However, the techniques adopted by Infection Monkey are very similar to conventional penetration testing, hence it differs from chaos engineering. There are no safety guarantees like roll-back, black-box testing techniques are employed and it is not based on experimentation as defined in the principles of chaos engineering. Our work is purely based on those principles and therefore employs a different philosophy to cloud security. Moreover, Infection Monkey targets cloud network layers, we target the cloud APIs, cloud account components e.g. users, ACPs etc. To the best of our knowledge, there is no other work that injects security faults against cloud systems.

VII. FUTURE WORK

A more intelligent recovery strategy will be implemented, that specifically takes note of the cloud assets that were changed during the security fault injection campaign. It is envisaged that this will improve efficiency by reducing the time overhead. Also in order to improve the performance and reduce the overhead due to network issues, it will be nice to implement the CloudStrike using serverless functions such as AWS Lambda. We did not analyze the performance of the Attack Graph construction in this article, however, this is planned as a future investigation. Furthermore, whilst we focused on IAM and cloud storage in this work, it will be interesting to extend it to cover other cloud services and systems such as AWS EC2 and Kubernetes.

VIII. CONCLUSION

We have presented *CloudStrike*, a security chaos engineering system designed for multi-cloud security. The *state-of-the-art* chaos engineering systems focus on detecting non-security weaknesses, which are largely based on *availability* properties. CloudStrike however, extends the gains of chaos engineering to security by injecting security faults that impact confidentiality, integrity and availability into cloud infrastructure. The notion of RDFI has been proposed to aid automatic, risk-based mechanisms by leveraging attack graph techniques and scoring detected vulnerabilities with CVSS algorithms. The security faults are realistic and are automatically injected using techniques that guarantee safety through *state reversibility* while verifying defined security properties. These security properties are specified as *security hypotheses* which are then proved. In order to trans-

[15]https://www.guardicore.com/infectionmonkey/

fer the output of the fault injections in an effective manner, we have adapted the MAPE-K framework and implemented the core functionalities as components of CloudStrike. These proposed methods are suitable for detecting vulnerabilities in cloud infrastructure, including human errors and misconfigurations, thereby enhancing cloud customer's confidence that such systems will withstand attacks in production e.g. the recurring AWS S3 data breaches. CloudStrike has been implemented and used for extensive evaluations against cloud infrastructure deployed on AWS and GCP.

REFERENCES

[1] *Top threats to cloud computing the egregious 11*, Cloud Security Alliance, Seattle, WA, USA, 2019.
[2] *2019 Cost Of a Data Breach Report*, Ponemon, Michigan, MN, USA, 2019.
[3] N. MacDonald, "Innovation insight for cloud security posture management," *Gartner Res.*, 2019.
[4] M. Ali, S. U. Khan, and A. V. Vasilakos, "Security in cloud computing: Opportunities and challenges," *Inf. Sci.*, vol. 305, pp. 357–383, Jun. 2015.
[5] *The Cloud Talent Drought Continues and is Even Larger Than you Thought*. Accessed: Jun. 5, 2020. [Online]. Available: https://www.forbes.com/sites/emilsayegh/2020/03/02/the-2020-cloud-talen%t-drought-is-even-larger-than-you-thought/
[6] Netflix. *Chaos Engineering Upgraded*. Accessed: Jul. 2, 2020. [Online]. Available: https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa
[7] A. Tseitlin, "The antifragile organization," *Queue*, vol. 11, no. 6, pp. 20–26, 2013.
[8] Netflix. *Netflix Chaos Monkey Upgraded*. Accessed: Jul. 2, 2020. [Online]. Available: https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d
[9] Codecentric. *Chaos Monkey for Spring Boot*. Accessed: Jul. 2, 2020. [Online]. Available: https://codecentric.github.io/chaos-monkey-spring-boot/
[10] L. Zhang, B. Morin, P. Haller, B. Baudry, and M. Monperrus, "A chaos engineering system for live analysis and falsification of exception-handling in the JVM," 2018, *arXiv:1805.05246*. [Online]. Available: http://arxiv.org/abs/1805.05246
[11] R. Miles, *Learning Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation*. Newton, MA, USA: O'Reilly Media, 2019.
[12] Bloomberg. *Powerfulseal*. Accessed: Jul. 2, 2020. [Online]. Available: https://www.techatbloomberg.com/blog/powerfulseal-testing-tool-kubernet%es-clusters/
[13] *Chaosmesh*. Accessed: Jul. 2, 2020. [Online]. Available: https://www.chaosmesh.com/
[14] A. Rinehart. *Chaoslingr*. Accessed: Jul. 2, 2020. [Online]. Available: https://github.com/Optum/ChaoSlingr
[15] K. A. Torkura, M. I. H. Sukmana, F. Cheng, and C. Meinel, "Security chaos engineering for cloud services: Work in progress," in *Proc. IEEE 18th Int. Symp. Netw. Comput. Appl. (NCA)*, Sep. 2019, pp. 1–3.
[16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, Mar. 2016.
[17] C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones, and A. Basiri, *Chaos Engineering—Building Confidence in System Behavior Through Experiments*. Newton, MA, USA: O'Reilly Media, 2017.
[18] A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating chaos experiments in production," 2019, *arXiv:1905.04648*. [Online]. Available: http://arxiv.org/abs/1905.04648
[19] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 57–66.
[20] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.
[21] *CIS Amazon Web Services Foundation*. Center for Internet Securityy, East Greenbush, MY, USA, 2018.
[22] *CIS Benchmarks for Google Cloud Platform*. Center for Internet Security, East Greenbush, MY, USA, 2018.
[23] *Cloud Penetration Testing Playbook*, CSA, New Delhi, Delhi, 2019.
[24] C. Rosenthal and N. Jones, *Chaos Engineering—System Resiliency in Practice*. Newton, MA, USA: O'Reilly Media, 2020.
[25] R. McLean. (2019). *A Hacker Gained Access to 100 Million Capital One Credit Card Applications and Accounts*. [Online]. Available: https://edition.cnn.com/2019/07/29/business/capital-one-data-breach/ind
[26] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Sec. Comput.*, vol. 1, no. 1, pp. 11–33, Jan./Mar. 2004.
[27] A. Avizienis, *Fundamental Concepts of Dependability*. Newcastle upon Tyne, U.K.: Univ. Newcastle upon Tyne, 2001.
[28] M. Al-Kuwaiti, N. Kyriakopoulos, and S. Hussein, "A comparative analysis of network dependability, fault-tolerance, reliability, security, and survivability," *IEEE Commun. Surveys Tuts.*, vol. 11, no. 2, pp. 106–124, 2nd Quart., 2009.
[29] L. Simoncini, "Resilient computing: An engineering discipline," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, p. 1.
[30] J.-C. Laprie, "From dependability to resilience," in *Proc. 38th IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2008, pp. 8–9.
[31] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State transition analysis: A rule-based intrusion detection approach," *IEEE Trans. Softw. Eng.*, vol. 21, no. 3, pp. 181–199, Mar. 1995.
[32] *Amazon Cloudwatch—Observability of Your AWS Resources and Applications on AWS and On-Premises*. Accessed: Jul. 2, 2020. [Online]. Available: https://aws.amazon.com/cloudwatch/
[33] *Aws X-ray Analyze and Debug Production, Distributed Applications*. Accessed: Jul. 2, 2020. [Online]. Available: https://aws.amazon.com/xray/
[34] K. A. Torkura, M. I. H. Sukmana, T. Strauss, H. Graupner, F. Cheng, and C. Meinel, "CSBAuditor: Proactive security risk analysis for cloud storage broker systems," in *Proc. IEEE 17th Int. Symp. Netw. Comput. Appl. (NCA)*, Nov. 2018, pp. 1–10.
[35] K. A. Torkura, M. I. H. Sukmana, M. Meinig, A. V. D. M. Kayem, F. Cheng, H. Graupner, and C. Meinel, "Securing cloud storage brokerage systems through threat models," in *Proc. IEEE 32nd Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, May 2018, pp. 759–768.
[36] P. Mell, K. Scarfone, and S. Romanosky, "A complete guide to the common vulnerability scoring system version 2.0," in *Proc. FIRST-Forum Incident Response Secur. Teams*, 2007, p. 23.
[37] K. Scarfone and P. Mell, "The common configuration scoring system (ccss): Metrics for software security configuration vulnerabilities," NIST, Gaithersbur, MD, USA, Tech. Rep. 7502, 2010.
[38] R. Wirtz and M. Heisel, "CVSS-based estimation and prioritization for security risks," in *Proc. 14th Int. Conf. Eval. Novel Approaches Softw. Eng.*, 2019, pp. 297–306.
[39] A. Continella, M. Polino, M. Pogliani, and S. Zanero, "There's a hole in that bucket!: A large-scale analysis of misconfigured S3 buckets," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 702–711.
[40] P. Engebretson, *The Basics Hacking Penetration Testing: Ethical Hacking Penetration Test. Made Easy*. Amsterdam, The Netherlands: Elsevier, 2013.
[41] L. Feinbube, L. Pirl, and A. Polze, "Software fault injection: A practical perspective," in *Dependability Engineering*. London, U.K.: IntechOpen, 2017.
[42] C. Alliance, *Cloud Penetration Testing Playbook*. Seattle, WA, USA: Cloud Security Alliance, 2019.
[43] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Lead. Issues Inf. Warfare Secur. Res.*, vol. 11, p. 80, Apr. 2011.
[44] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, "An attack graph-based probabilistic security metric," in *Proc. Annu. Conf. Data Appl. Secur. Privacy*, 2008, pp. 283–296.
[45] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proc. ACM Int. Conf. Manage. Data*, 2015, pp. 331–346.
[46] P. Alvaro and S. Tymon, "Abstracting the geniuses away from failure testing," *Queue*, vol. 15, no. 5, pp. 29–53, Oct. 2017.
[47] *AWS Lambda; Run Code Without Thinking About Servers. Pay Only for the Compute Time you Consume*. Accessed: Jul. 2, 2020. [Online]. Available: https://aws.amazon.com/lambda/
[48] W. Du and A. P. Mathur, "Vulnerability testing of software system using fault injection," Purdue Univ., West Lafayette, Indiana, Tech. Rep. COAST TR02-98, 1998.

[49] J. Fonseca, M. Vieira, and H. Madeira, ''Evaluation of Web security mechanisms using vulnerability & attack injection,'' *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 5, pp. 440–453, Sep. 2014.
[50] S. Bruning, S. Weissleder, and M. Malek, ''A fault taxonomy for service-oriented architecture,'' in *Proc. 10th IEEE High Assurance Syst. Eng. Symp. (HASE)*, Nov. 2007, pp. 367–368.

KENNEDY A. TORKURA received the M.Sc. degree in cyber security from Lancaster University, U.K. He is currently pursuing the Ph.D. degree in cyber-security with the Internet Technologies and Systems Research Group, Hasso Plattner Institute for Digital Engineering, Potsdam, Germany. His research focuses on security risk and threat analysis of cloud-native environments, security chaos engineering, and incident response.

MUHAMMAD I. H. SUKMANA received the B.Sc. degree in information technology from the Asia Pacific University of Technology and Innovation, Malaysia, and the M.Sc. degree in communication systems and networks from the Technical University of Applied Science Cologne, Germany. He is currently pursuing the Ph.D. degree with the Hasso Plattner Institute, University of Potsdam, Germany. His current research interests include applied cryptography, cloud security management, and enterprise access control.

FENG CHENG received the B.Eng. degree from the Beijing University of Aeronautics and Astronautics, China, the M.Eng. degree from the Beijing University of Technology, China, and the Ph.D. degree from the University of Potsdam, Germany. He is currently a Senior Researcher Heading with the IT Security Engineering (Sec-Eng) Team, Hasso Plattner Institute (HPI), University of Potsdam. His research is mainly focused on big security data analytics, network security, firewall, IDS/IPS, attack modeling and penetration testing, SOA and cloud security, SDN, and so on.

CHRISTOPH MEINEL received the Ph.D. degree with Humboldt University, Germany. He is currently the President and CEO of the Hasso Plattner Institute, Germany, and a Full Professor with the Internet Technologies and Systems Chair, University of Potsdam, Germany. His research focuses on future internet technologies, in particular internet and information security, web 3.0, semantic-, social- and service-web, as well as on innovative internet applications, such as e-learning and telemedicine.